Project Design Document

February 25, 2025

1 Introduction

This document provides a comprehensive design overview of the Furniture Store Backend System. It describes the system architecture, key components, applied design patterns, and how the design evolved during the implementation phase.

The primary goal of this project is to create a robust and scalable e-commerce backend that supports user authentication, inventory management, a shopping cart, checkout functionality, and an order-processing system.

2 Initial System Design

Initially, the project was planned as a simple e-commerce backend to handle:

- User Authentication (Register, Login, Profile Management)
- CRUD operations for Furniture Management
- Inventory Management to track available stock
- Shopping Cart for customers to add furniture before purchase
- Order Processing and Checkout with Payment Handling

However, as development progressed, several improvements were introduced to enhance scalability, maintainability, and performance.

3 Design Evolution and Changes

- Implemented the strategy pattern for handling discounts. Instead of hardcoded discount conditions, we introduced an abstract class for discount strategies with percentage-based and fixed-amount discount implementations. This makes it easy to introduce new discount methods in the future.
- Used the factory pattern for user creation. Instead of handling user creation in multiple places, we centralized it inside a user manager class. This ensures all users are created consistently with hashed passwords and proper validation.
- Applied the observer pattern to automatically update stock and clear shopping carts when an order is placed. This prevents the need for manually handling these updates and ensures system consistency.
- Improved the checkout process by validating stock availability before confirming an order. This prevents users from ordering out-of-stock items and ensures inventory updates correctly.
- Implemented a payment simulation mechanism. While real payment integration is not included, the system now allows different payment methods and ensures valid selections.
- Added automated order status updates. Orders transition from pending to processed automatically when they are placed.
- Introduced role-based access control. Admins can update inventory and manage orders, while regular users can only interact with the shopping cart and checkout.

4 Final Architecture

The final architecture consists of the following key modules:

4.1 1. User Authentication

- Secure user registration and login
- Profile management
- Role-based access control (admin vs. customer)

4.2 2. Furniture Management

- CRUD operations for managing inventory
- Category-based organization (e.g., chairs, sofas, tables)
- Price calculations including tax and discounts

4.3 3. Shopping Cart

- Allows users to add/remove items
- Automatically calculates total cost

4.4 4. Order Processing and Checkout

- Handles payment validation
- Updates inventory and order status

4.5 5. Discount System

- Supports multiple discount strategies using the strategy pattern

5 Technology Stack

- Backend: Django Rest Framework (DRF)
- Database: SQLite
- Authentication: JWT-based authentication
- Testing: Pytest, pytest-cov for coverage tracking

6 Design Patterns Used

The project implements several design patterns to ensure modularity and maintainability.

6.1 1. Strategy Pattern (Used for Discounts)

Purpose: Allows the system to dynamically switch between different discount strategies.
Implementation:

- The ‘DiscountStrategy‘ class defines an interface for discount types.
- ‘PercentageDiscount‘ and ‘FixedAmountDiscount‘ implement this interface.
- The shopping cart applies a discount strategy dynamically.

Example:

```
class PercentageDiscount(DiscountStrategy):
    def __init__(self, percentage):
        self.percentage = percentage
    def apply_discount(self, original_price):
        discount_amount = (self.percentage / 100) * original_price
        return max(0, original_price - discount_amount)
```

6.2 2. Factory Pattern (Used for User Management)

Purpose: Encapsulates user creation logic, ensuring consistent password hashing and user validation.
Implementation:

- The 'CustomUserManager' class provides a 'create$_user$()'$method.Itensureseveryuserisproperlycreatedwithhashedpassu$
Example:

- ```
  class CustomUserManager(BaseUserManager):
      def create_user(self, username, email, password=None):
          user = self.model(username=username, email=email)
          if password:
              user.set_password(password)
          user.save(using=self._db)
          return user
  ```

6.3 3. Composite Pattern (Used for Shopping Cart and Orders)

Purpose: Allows an order to contain multiple order items, treating them as a single unit.
Implementation:

- The 'Order' model acts as a composite structure, containing multiple 'OrderItem' instances.
- The API manipulates orders as a whole rather than handling individual items.

Example:

```
class Order(models.Model):
    user = models.ForeignKey(CustomUser, on_delete=models.CASCADE)
    total_price = models.DecimalField(max_digits=10, decimal_places=2)
    created_at = models.DateTimeField(auto_now_add=True)
```

7 Testing and Code Coverage

7.1 Testing Framework

- Used pytest for unit and integration tests.
- Used pytest-cov to measure test coverage.
- Ensured at least 80% code coverage for backend logic and API endpoints.
```

7.2 Coverage Report

```
---------- coverage: platform win32, python 3.12.3-final-0 -----------
Name                                          Stmts   Miss  Cover
-----------------------------------------------------------------
api\__init__.py                                   0      0   100%
api\admin.py                                      6      0   100%
api\apps.py                                       4      0   100%
api\migrations\0001_initial.py                    6      0   100%
api\migrations\0002_shoppingcart.py               6      0   100%
api\migrations\0003_order_orderitem.py            6      0   100%
api\migrations\__init__.py                        0      0   100%
api\models\__init__.py                            5      0   100%
api\models\discount.py                           16      6    62%
api\models\furniture.py                          21      1    95%
api\models\inventory.py                          36     25    31%
api\models\order.py                              21      2    90%
api\models\shopping_cart.py                      13      1    92%
api\models\user.py                               37      2    95%
api\serializers\__init__.py                       5      0   100%
api\serializers\furniture.py                     12      0   100%
api\serializers\inventory.py                      7      0   100%
api\serializers\order.py                         14      0   100%
api\serializers\shopping_cart.py                 15      1    93%
api\serializers\user.py                          11      0   100%
api\tests.py                                      1      0   100%
api\tests\test_full_regression.py                43      0   100%
api\tests\test_furniture_api.py                  39      0   100%
api\tests\test_inventory_api.py                  69      0   100%
api\tests\test_shopping_cart.py                  18      0   100%
api\urls.py                                      15      0   100%
api\views\__init__.py                             6      0   100%
api\views\checkout.py                            32      3    91%
api\views\furniture.py                           20      8    60%
api\views\inventory.py                           67     27    60%
api\views\order.py                               20      8    60%
api\views\shopping_cart.py                       31     10    68%
api\views\user.py                                40     10    75%
-----------------------------------------------------------------
TOTAL                                           642    104    84%
```

Figure: Test coverage report generated using pytest –cov.

8 Conclusion

This document captures the architectural evolution and final implementation of the Furniture Store Backend System. By incorporating design patterns such as strategy, factory, and composite, the system is now modular, maintainable, and scalable.

Future improvements may include:

- Integrating real-world payment gateways.
- Adding order tracking functionality.
- Implementing real-time stock updates with WebSockets.